Методичка 9.4 - Знакомство с фреймворком radare2. Часть 3.

Анализ бинарного файла

В качестве жертвы была подготовлена программа prog-7.4.

Исходный код программы prog-7.4

```
#include <stdio.h>
#include <string.h>

void rot13 (char *s) {
  if (s == NULL) { return; }

  for (int i = 0; s[i]; i++) {
    if (s[i] >= 'a' && s[i] <= 'm') { s[i] += 13; continue; }
    if (s[i] >= 'A' && s[i] <= 'M') { s[i] += 13; continue; }
    if (s[i] >= 'n' && s[i] <= 'z') { s[i] -= 13; continue; }
    if (s[i] >= 'N' && s[i] <= 'Z') { s[i] -= 13; continue; }
  }
}

int check_key(char *key) {
  char tmp_key[] = "superkey";
  rot13(tmp_key);

  char tmp[32];
  strcpy(tmp, key);
  return strcmp(tmp_key, tmp);
}

int main(int argc, char *argv[]) {

  char buff[32];
  memset(buff, 0, sizeof(buff));

  printf("\nInsert Key: ");
  scanf("%s", buff);

  if (check_key(buff) != 0) {
    printf("\nERROR. License key is incorrect.\n\n");
    return 0;
  }
```

```
  printf("\nCongratulations! License key is correct.\n\n");
  return 0;
}
```

Компиляция программы:

```
$ gcc prog-7.4.c -o prog-7.4 -fno-stack-protector -z execstack
```

Запуск.

```
$ ./prog-7.4

Insert Key: fhcrexrl

Congratulations! License key is correct.
```

Прежде чем приступить к анализу бинарного файла, его стоит проверить на наличие встроенных механизмов защиты с помощью утилиты rabin2. Для нас важно убедиться, что механизмы защиты NX Bit и Stack Canary неактивны для данного файла.

```
$ rabin2 -I prog-7.4
...
canary   false
nx       false
...
```

Видим, что механизм Stack Canary и NX Bit отключены. Более подробно эти механизмы разобраны в курсе “Бинарные уязвимости”.

Если говорить вкратце, то механизм защиты Stack Canary затрудняет эксплуатацию уязвимостей переполнения буфера в стеке, а механизм защиты NX Bit препятствует исполнению данных в адресном пространстве процесса. В программе prog-7.4 они выключены и нас не побеспокоят.

Давайте запустим программу и посмотрим, что происходит при запуске.

```
$ ./prog-7.4

Insert Key: blabla

ERROR. License key is incorrect.
```

Видим, что теперь приложение принимает ключ через пользовательский ввод во время работы приложения, а не как раньше, через входные аргументы.

Обнаружение уязвимости

Давайте попробуем отладить программу средствами radare. Запустим нашу программу в режиме debug.

$ r2 -d prog-7.4

Выполняем автоматический анализ.

[0xb7f2ca20] aaa

Далее продолжаем выполнение программы, пока не дойдем до функции main. Для этого мы можем воспользоваться командой - dcu main (debug continue until).

[0xb7f2ca20] dcu main

Переходим в режим графов с помощью команды VV. Получаем дизассемблированный код функции main.

Видим, что после того, как программа просит нас ввести ключ, происходит вызов функции scanf.

Функция scanf выполняет копирование данных из stdin в буфер фиксированной длины.
Это серьезная ошибка. Если передать данных больше, чем размер буфера, то буфер переполнится. Это очень похоже на наличие уязвимости.

Разработка PoC

После того, как нам показалось, что мы нашли уязвимость в коде, нам необходимо проверить наличие уязвимости и провести эксперимент. Для того, чтобы передать данных больше, чем размер буфера, нам нужно знать размер буфера.

Мы можем конечно вернуться в функцию main и посмотреть сколько байт выделялось для нашего буфера, но мы пойдем по более сложному пути, так как не всегда есть возможность легко узнать размер буфера, особенно, если он выделяется в куче и происходит это динамически.

Для создания специальной последовательности байт мы будем использовать утилиту из фреймворка radare2, которая называется ragg2.

$ ragg2 -P 100 -r
AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAAPAAQAARAASAATAAUAAVAAWAAXAAYAAZAAaAAbAAcAAdAAeAAfAAgAAh

Через параметр -P можно указать размер последовательности, а через параметр -r мы просим печатать символы в явном виде.

Теперь у нас есть последовательность байт и мы можем проверить наше приложение.

$ ./prog-7.4

Insert Key:
AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAAPAAQAARAASAATAAUAAVAAWAAXAAYAAZAAaAAbAAcAAdAAeAAfAAgAA
Segmentation fault (core dumped)

Мы видим, что программа завершилась с ошибкой, но какой размер буфера всё еще не знаем. Знаем только, что он меньше 100.

Для того, чтобы узнать размер буфера нам потребуется другая утилита из фреймворка - это утилита rarun2. Эта утилита умеет перезаписывать файловый дескриптор, например, stdin. Это очень удобно, когда вам нужно запустить программу с длинными аргументами или передать огромное количество данных в stdin.

Записываем нашу последовательность в файл, чтобы потом можно было к ней обратиться.

$ ragg2 -P 100 -r > tmp.data

Затем создаем конфигурационный файл для утилиты rarun2.

```
$ cat profile.rr2
#!/usr/bin/rarun2
stdin=./tmp.data
```

Запускаем нашу программу в режиме debug совместно с утилитой rarun2.

```
$ r2 -r profile.rr2 -d prog-7.4
Process with PID 3130 started...
= attach 3130 3130
bin.baddr 0x08048000
Using 0x8048000
asm.bits 32
glibc.fc_offset = 0x00148
 -- Help subcommand will be eventually removed.
[0xf7f50c70]>
```

Перезапускаем, чтобы передать данные в stdin.

```
[0xf7f50c70]> dc

    Insert Key:
    child stopped with signal 11
    [+] SIGNAL 11 errno=0 addr=0x41417641 code=1 ret=0
    [0x41415341]>
```

Мы запустили нашу программу, передали содержимое tmp.data в stdin с помощью rarun2 и получили SIGSEV 11.

Вы заметили, что адрес в консоли изменился? Байт 0x41 это байт ‘A’, а байт 0x64 это байт ‘d’, таким образом адрес сейчас соответствует строке “AAdA”. Это значит, что мы перетерли адрес возврата.

Мы до сих пор не знаем размер буфера. Фреймворк radare2 позволяет нам найти смещение в заданном значении шаблона де Брёйна. Для этого воспользуемся следующей командой

```
[0x41417641]> wopO `dr eip`
85
```

Теперь мы знаем, что перезапись адреса возврата происходит после 85 байт, а как это делать вы уже узнаете из курса, который посвящен бинарным уязвимостям.